|  |
| --- |
| Term Project Evaluation Form CSCE614 |

Name: Biren Parmar & Shanmathi Mookiah

Project Title: IMPROVING CACHE PERFORMANCE BY EXPLOITING

READ-WRITE DISPARITY

Date & Time this report submitted: 3 May 2017, 17:00 hrs

|  |  |  |
| --- | --- | --- |
| **Evaluation** | **MAX. Score** | **Your Score** |
| Overall Organization Title, abstract (problem attempted, outline of your results, improvements) table of contents, introduction (general description of the problem, motivation, related work, and goals), description of the problem and related definitions and background, description of your work, experimental results, conclusions, appendices (your code may be included here), tables, and figures. | 10 |  |
| In depth description of the problem and its significance | 10 |  |
| Techniques used and definitions (should be self-sufficient) | 20 |  |
| Description of your work and results | 40 |  |
| Technical evaluation (soundness of approach and depth) of results | 60 |  |
| Conclusions, summary of work, and directions for further work | 15 |  |
| Appendices (if applicable) | 5 |  |
| References | 10 |  |
| Overall quality of report | 30 |  |
|  | | |
| **Total** | 200 |  |

CSCE 614 – COMPUTER ARCHITECTURE

TERM PROJECT REPORT

SPRING 2017

IMPROVING CACHE PERFORMANCE BY EXPLOITING

READ-WRITE DISPARITY

Submitted by Biren Parmar & Shanmathi Mookiah

1. **ABSTRACT**

A memory request taking more than hundreds of cycles to access the memory is the major bottleneck, impacting the performance gap between the memory and processor, in microprocessor design. This performance gap motivates the improvement in the cache design and management. The cache lines that serve the loads are more critical compared to the cache lines that serve the stores. Traditional cache management mechanisms do not consider the disparity between read-write criticality. The new idea in this paper [1] for our project work distinguishes between lines that are reused by reads versus those that are reused only by writes. A Read-Write Partitioning (RWP) policy that minimizes read misses by dynamically partitioning the cache into clean and dirty partitions, where partitions grow if they are more likely to receive future read requests. This proposed cache management takes this disparity into account and will minimize the read misses and provides better performance than the baseline LRU cache management policy.

TABLE OF CONTENTS

[1. ABSTRACT 3](#_Toc481592083)

[2. OBJECTIVE 6](#_Toc481592084)

[The key objectives of the project are 6](#_Toc481592085)

[3. Introduction 7](#_Toc481592086)

[4. Read Write Partitioning 11](#_Toc481592087)

[4.1 RWP Framework 12](#_Toc481592088)

[4.2 Predicting partition sizes 13](#_Toc481592089)

[5. Evaluation methodology 15](#_Toc481592090)

[5.1 Baseline configuration 15](#_Toc481592091)

[5.2 Benchmarks 15](#_Toc481592092)

[5.3 Simulated Cache Configurations 17](#_Toc481592093)

[6. Results /performance analysis 18](#_Toc481592094)

[6.1 Miss rate 18](#_Toc481592095)

[6.2 Replacement rate 21](#_Toc481592096)

[6.3 Memory traffic 24](#_Toc481592097)

[6.4 AMAT 27](#_Toc481592098)

[6.5 Storage overhead 28](#_Toc481592099)

[6.6 Summary 28](#_Toc481592100)

[7. Conclusions and future work 29](#_Toc481592101)

[8. Contrasting: PAPER [1] and PROJECT 30](#_Toc481592102)

[9. References 31](#_Toc481592103)

[10. Appendix A 32](#_Toc481592104)

[11. Appendix B 37](#_Toc481592105)

1. **OBJECTIVE**

The key objectives of the project are

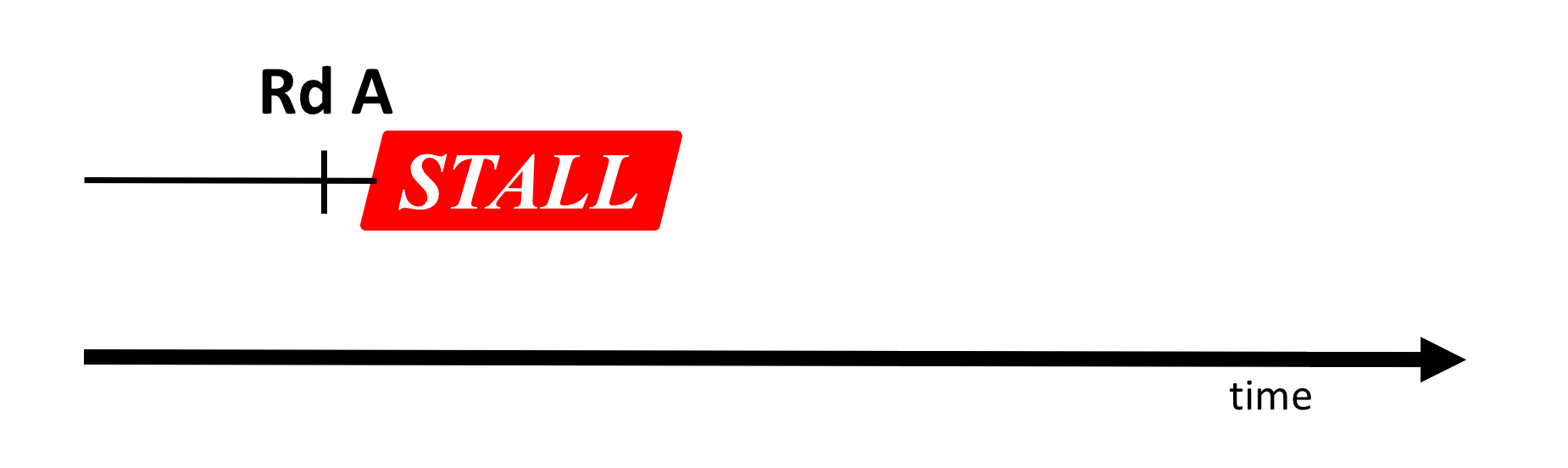
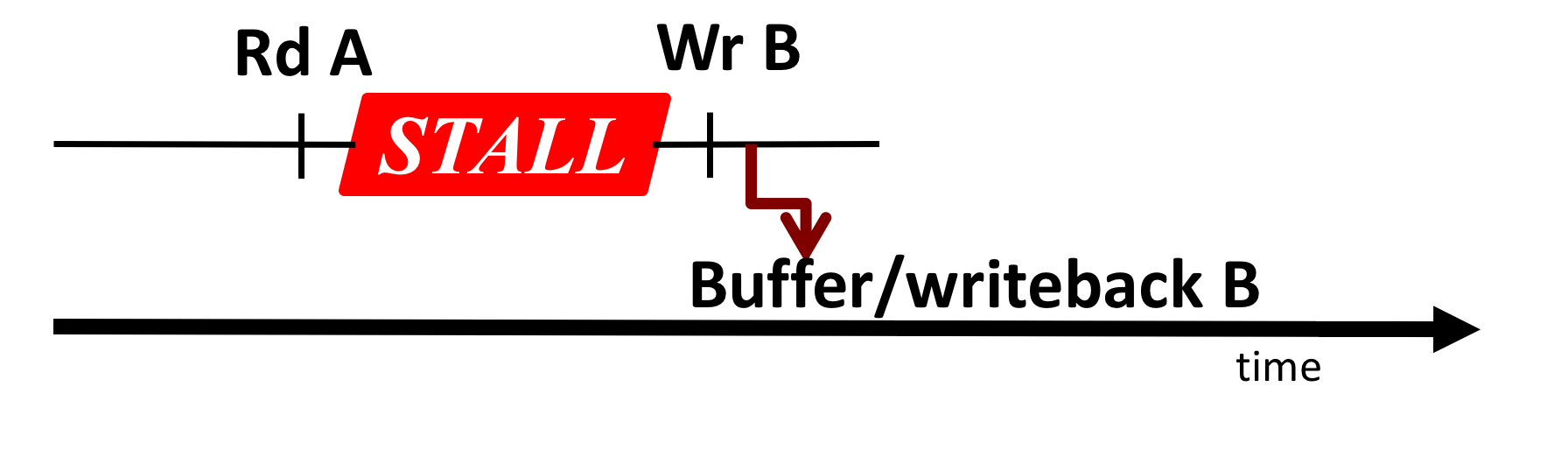
* Understanding through data the criticality and locality, i.e. cache lines that read versus the cache lines that are only written.
* Exploitation of the disparity between the read and the write criticality using the proposed RWP, to divide the LLC into two logical partitions for dirty and clean lines. RWP predicts the best partition sizes to increase the likelihood of the future read hits.
* Comparison of RWP with the prior state of the art cache management policies.
* Implementation of the RWP in simplescalar and analyzing the results of benchmarks of the SPEC CPU2000 suite.
* Obtaining performance improvement close to the paper [1]. Miss rate reduction by 10%

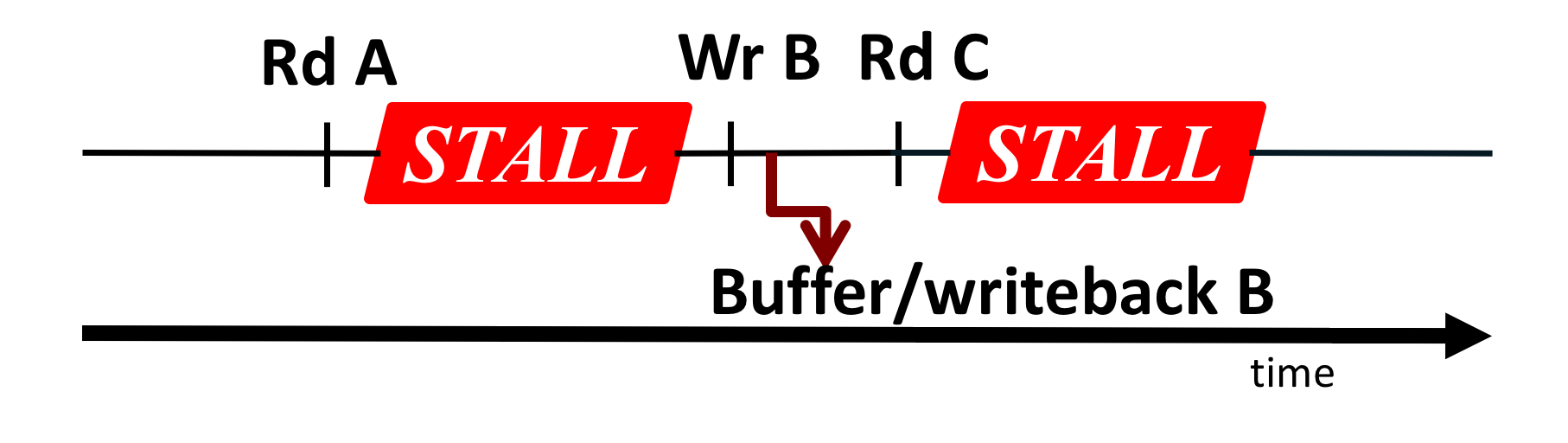
and memory traffic reduction by 20 %.

1. **Introduction**

In most of the cases, the cache lines that serve the loads are more critical compared to the cache lines that serve the stores. The primary rationale for this is that cache read can cause the stalls in the processor execution while in contrast the most cache write can be buffered in the store buffer. The criticality between the loads and store in the processor pipeline is treated differently, giving higher attention to the latency of the loads while the stores are buffered in the store buffers. The associated read and write in cache hierarchy does not distinguish as in the same as the processor pipeline does, even though the read request latency is often more critical than the store latency.

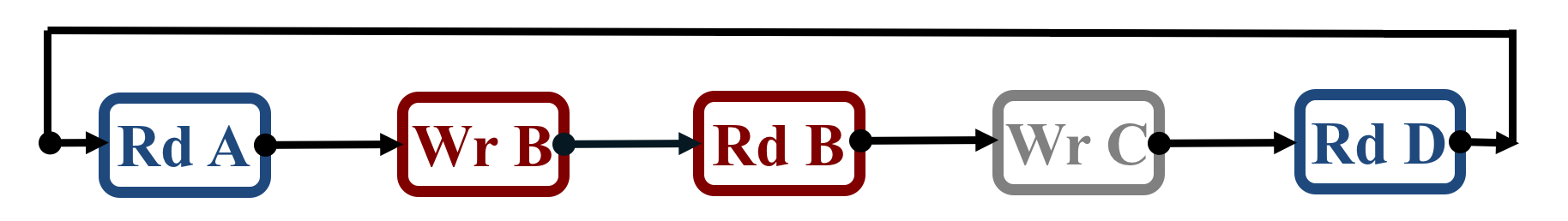
If there are no independent instructions to execute, a read miss can stall the processor. A miss at read A stalls the processor. Writes are usually buffered and not critical. So, miss at Wr B does not stall the processor. Rd C again stalls the processor and so on.



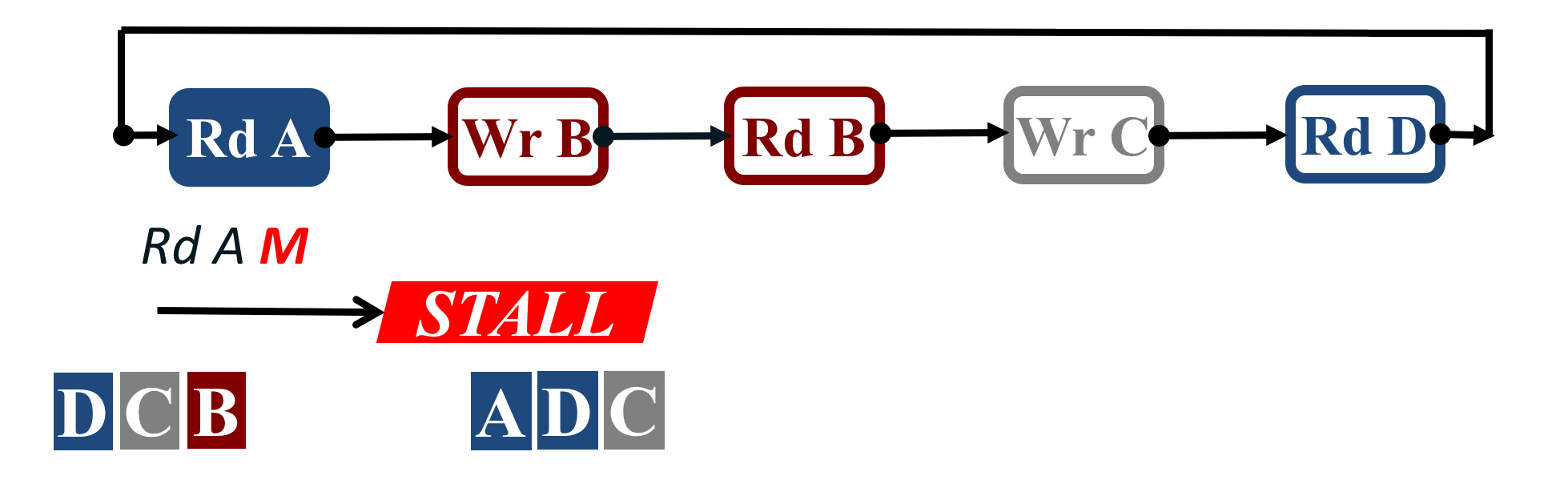


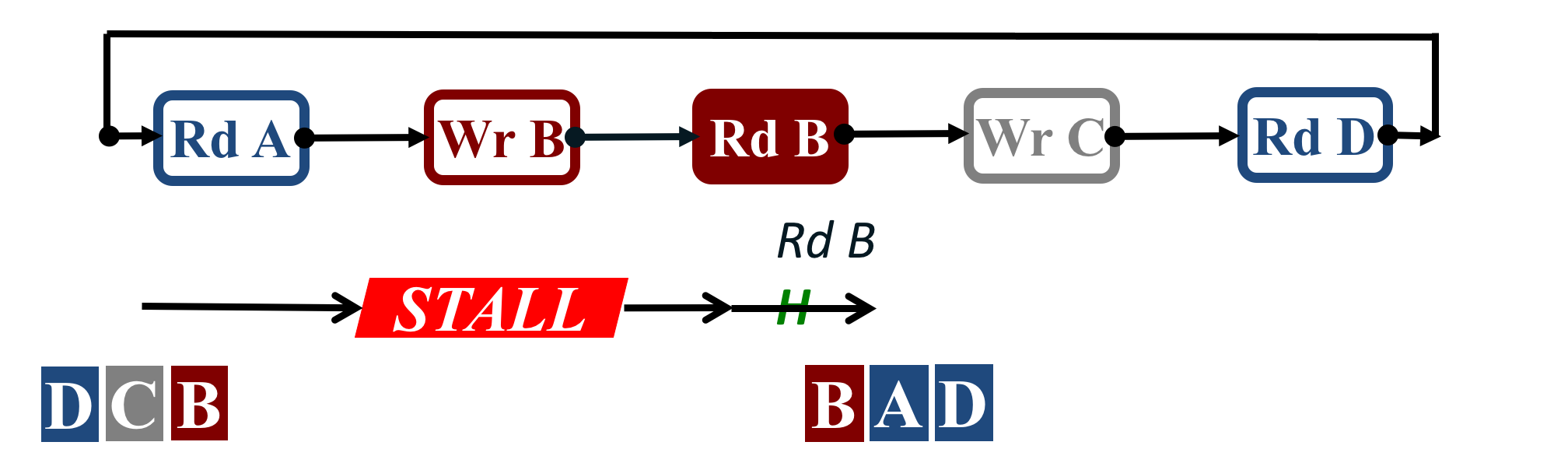
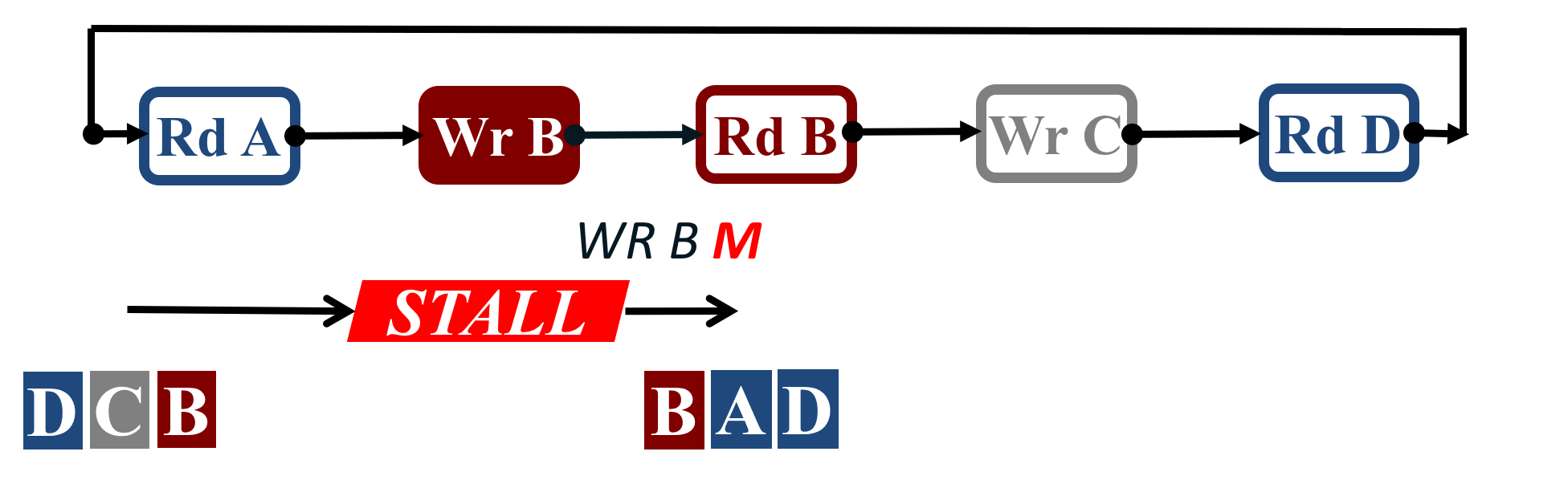
The traditional cache management does not take the read-write disparity into account. The proposed cache management takes this disparity into account and will minimize the read misses and provides better performance than the baseline LRU cache management policy.

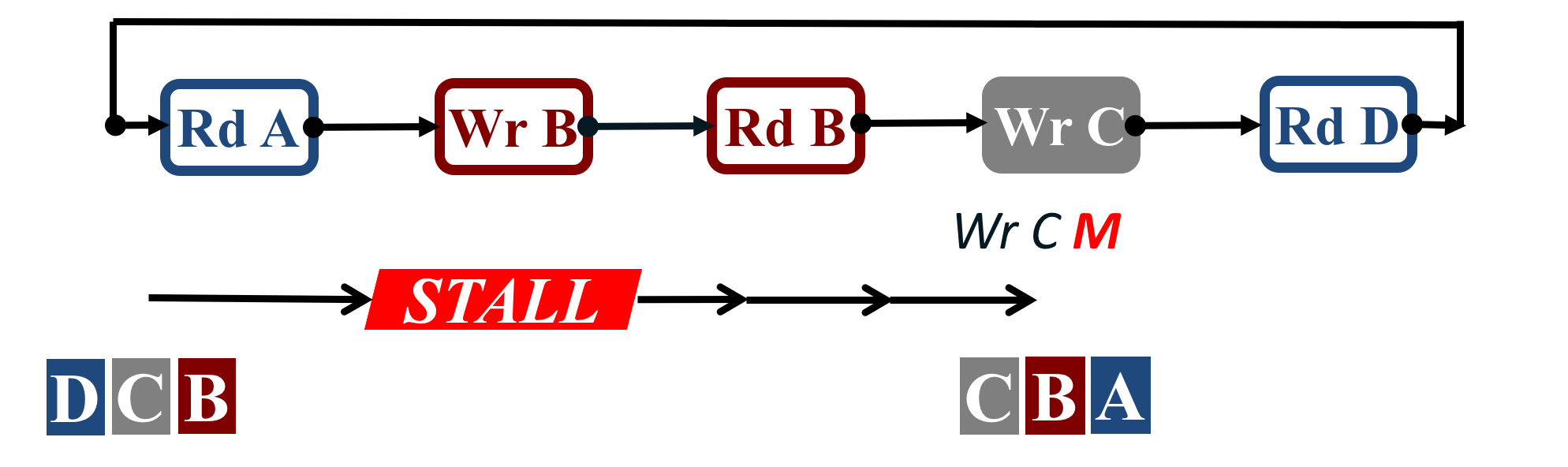
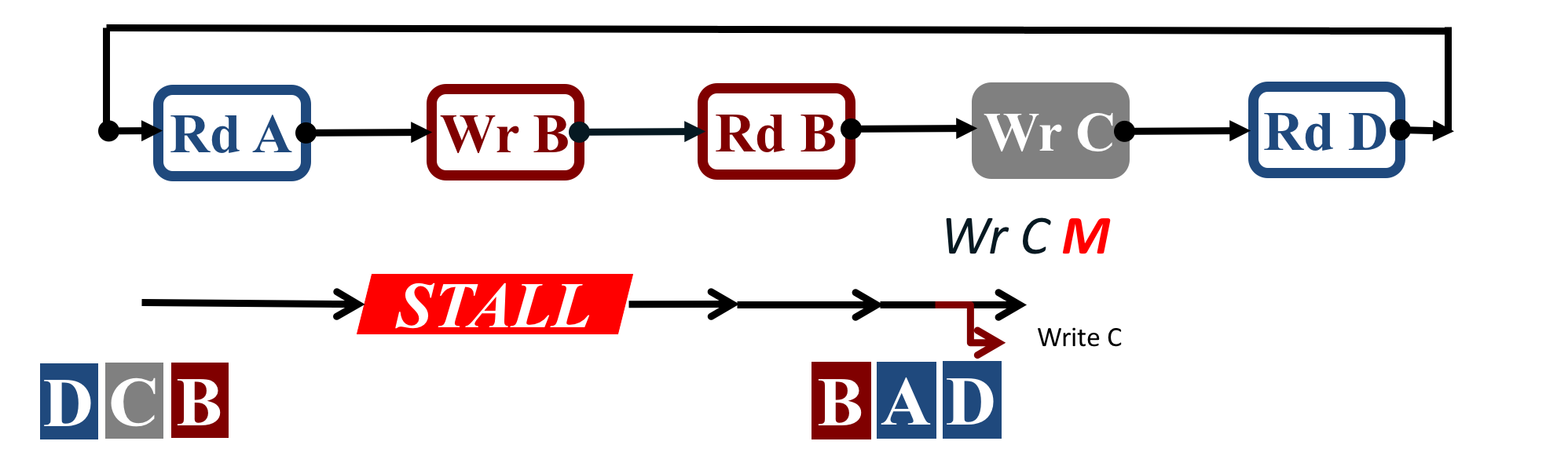
Let’s see an example that shows how this idea can improve the performance. Here we have a loop, that touches four cache lines, A, B, C and D. A and D are read only lines, where B is both read and written and C is only written to.

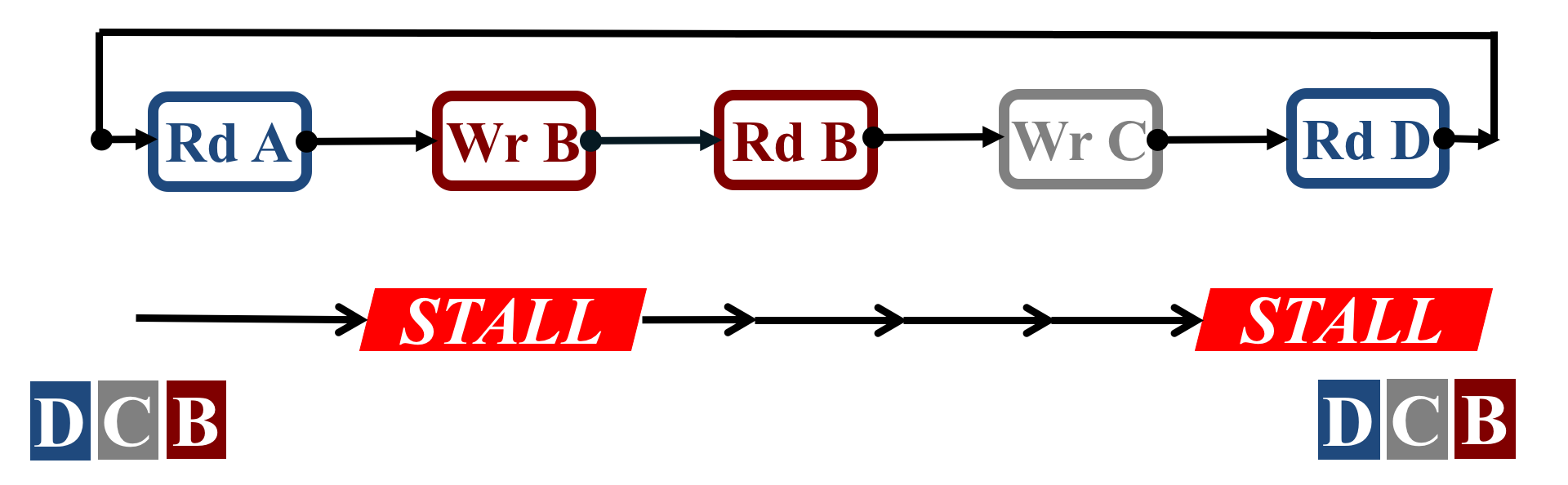
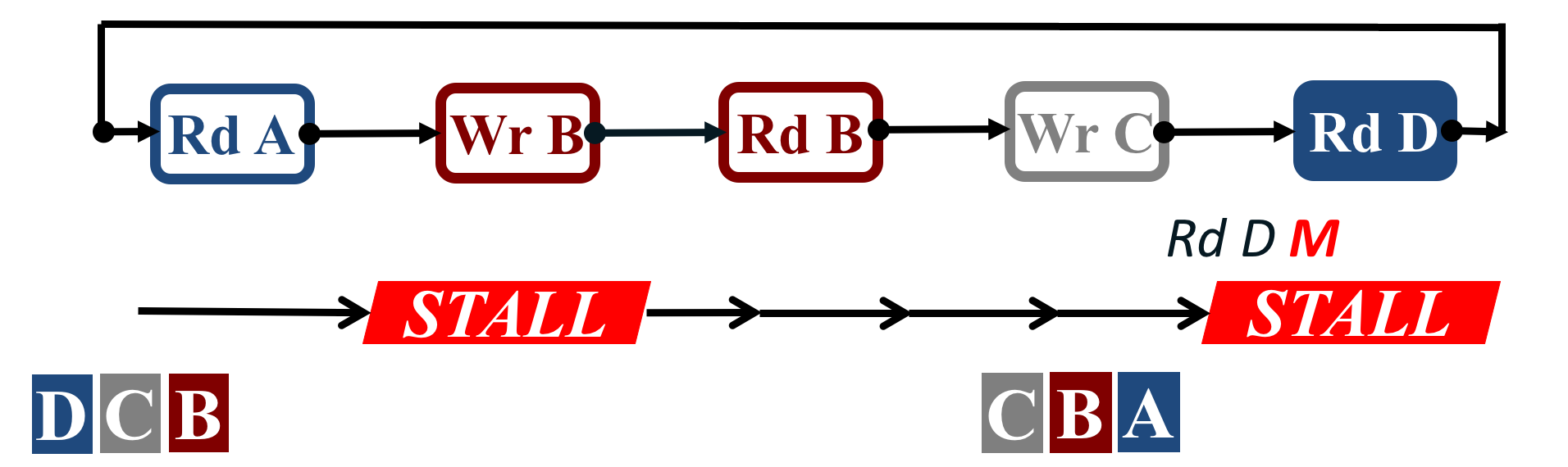


First let’s see what happens in the LRU replacement policy. Suppose we have a cache that can hold only three lines. Since the loop touches four block, it thrashes the cache. In a stable iteration, Load A causes a miss, as the last access in the previous iteration replaced A. and it stalls the processor. The next access to B is also a miss, but does not cause a stall as the write gets buffered. Next, we access B again and since it is already in the cache, it is a hit. Next access to C also causes a miss, but not one the critical path. Last access is to D, load D is also a miss, as it got evicted by C. So, there are two stalls per iteration for this policy.

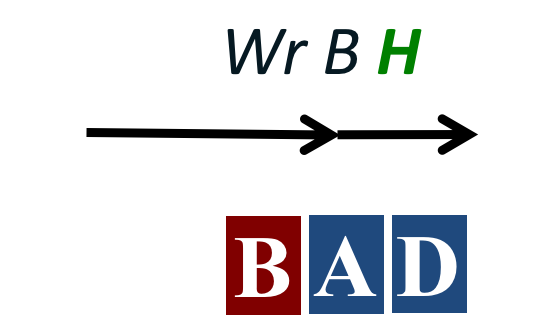
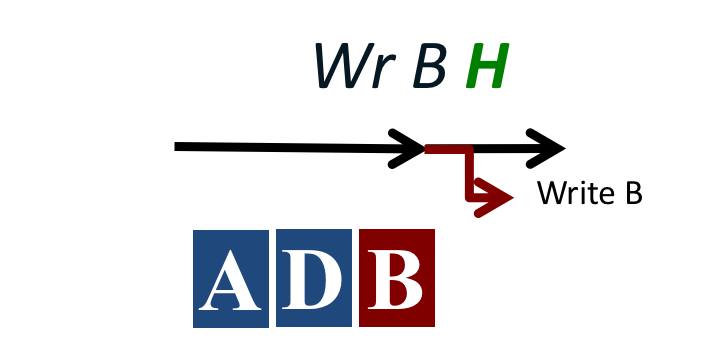
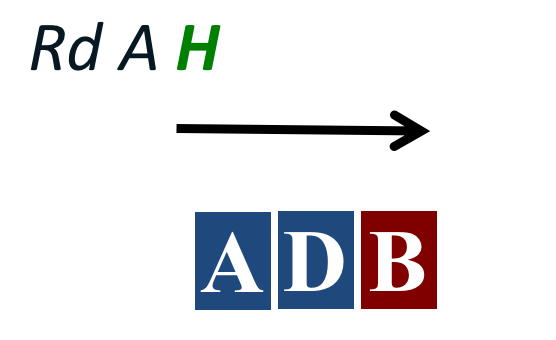
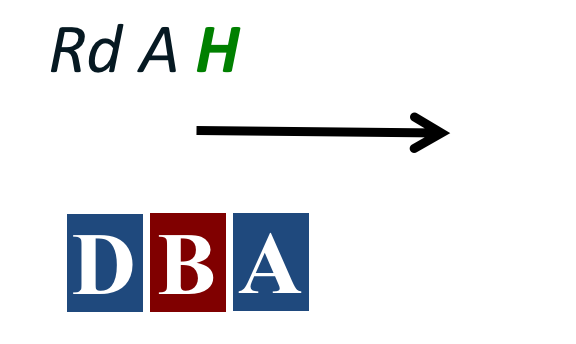


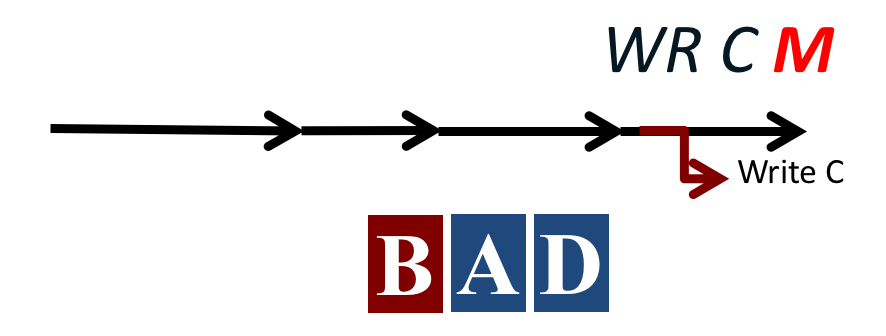
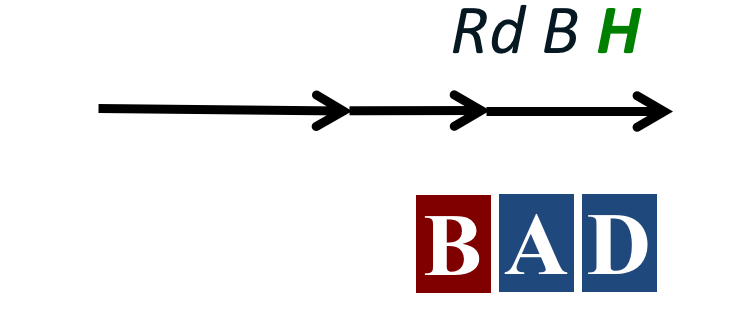
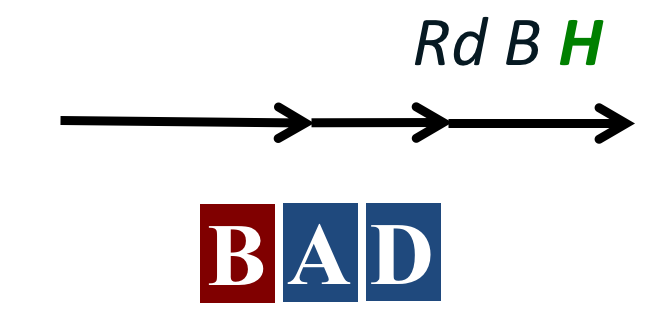


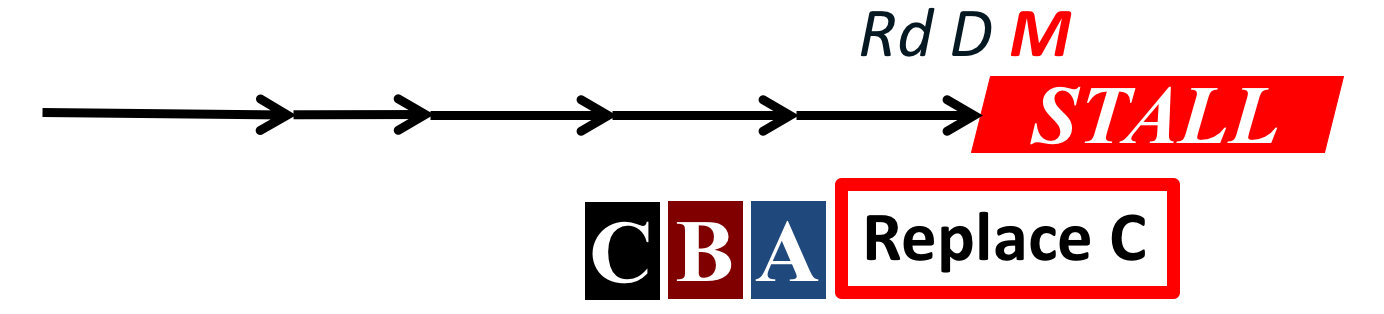
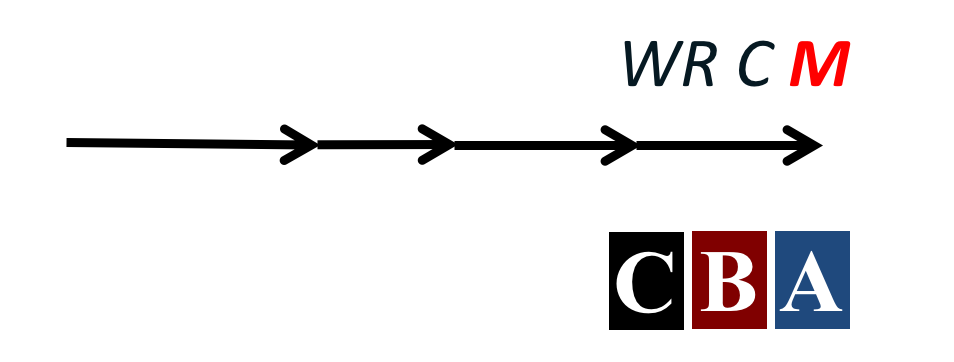


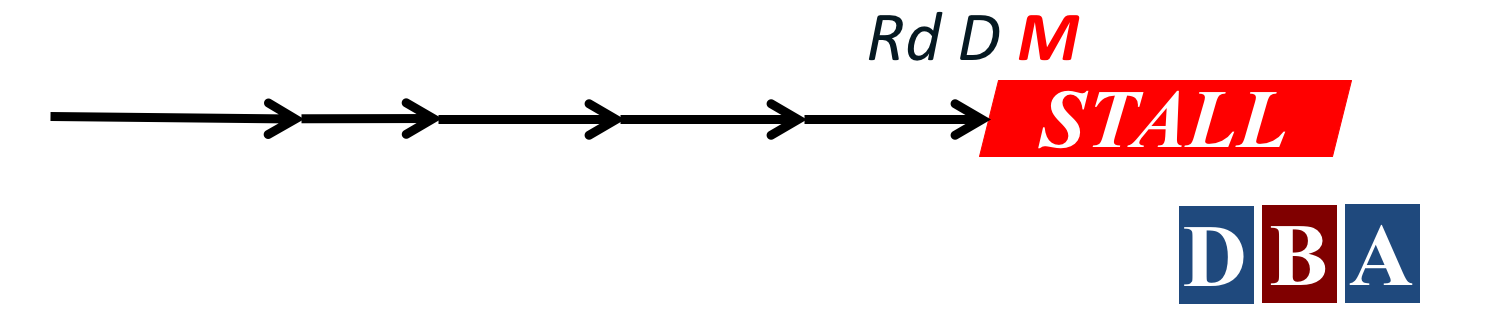


Now let’s see what happens in a read-biased policy. Since A, B, and D service read requests, this policy tries to protect these three lines. In a stable iteration, first access to A becomes a hit, as it is protected in the previous iteration. Accesses to B, and C becomes a hit. When we access D, we encounter a miss, and instead of replacing the LRU block, we replace C, as it does not service any read requests. So by protecting A we eliminate a miss. Read-biased policy has 1 stall per iteration where LRU have 2 stalls per iteration. So, evicting write-only lines can improve performance.









So there is only one stall per iteration in this case.

The interesting thing here is, we cannot just evict dirty lines. Both B and C are dirty lines, but B is protected because it is read, where C is not protected as it is only written to. A no-write allocate policy would not differentiate between them and will bypass both B and C, but this will result in a miss when B is read. So, we argue that we cannot treat all the dirty lines in the same way.

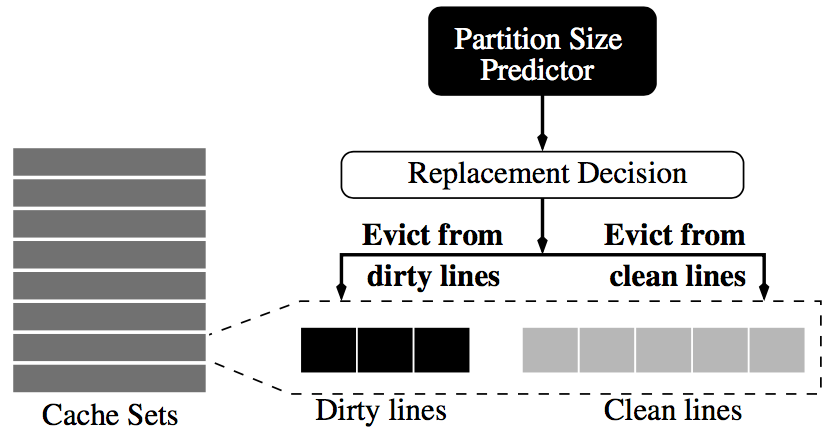
In the term project, a novel cache management called, Read Write Partitioning (RWP) is implemented referring to [1][4][3]. RWP considers the disparity of criticality of latencies that exist between the read and write cache accesses, and minimizes the read misses by dynamically partitioning the cache in two partitions, clean and dirty partitions. The size of partitions grows if they are more likely to receive future read request. RWP is based on the argument that caches should be designed to favor the read request over the less critical write requests. This cache management increases the probability of the cache hits for critical read requests at the cost of the causing the less critical write request to miss.

1. **Read Write Partitioning**

Some applications have more read requests to clean lines, whereas other applications have more read requests to dirty lines. Thus, in RWP, the cache is logically divided into two partitions, a clean or read partition and a dirty or write partition utilizing disparity in read reuse in clean and dirty lines. The RWP has the following actions to happen in the implementation.

1. Partition the cache into clean and dirty lines
2. Predict the partition size that maximizes read hits
3. Maintain the partition through replacement by using Dynamic insertion policy

The following picture explains the three main actions requisite to the implementation of RWP.

****

* 1. RWP Framework

The cache is divided into logical partitions for the clean and write lines. The dirty bit value decides if it is a dirty and clean line and decides which group it belongs to. So, when a write happens to a clean line, the dirty bit is set to 1 and then logically the line is considered to be the part of the dirty partition. When a new line is allocated, using the Dynamic insertion policy, the new line is inserted to the dirty or clean partition based on the dirty bit. For a replacement in the cache line, there can be three cases to consider:

1. Current number of dirty lines is greater than the predicted the dirty partition size.

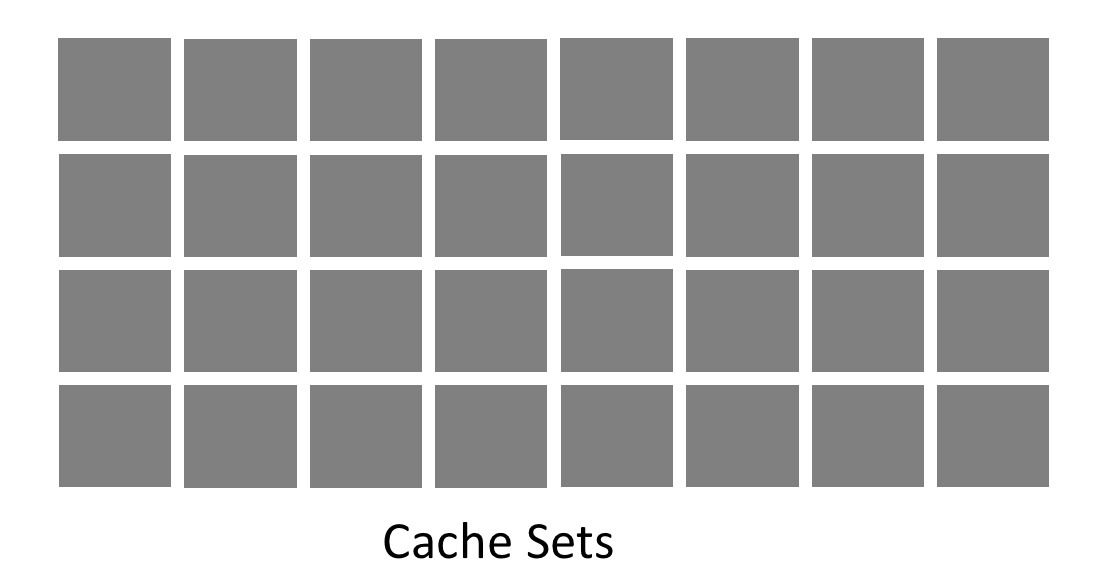
RWP picks the LRU line from the dirty partition as the replacement victim.

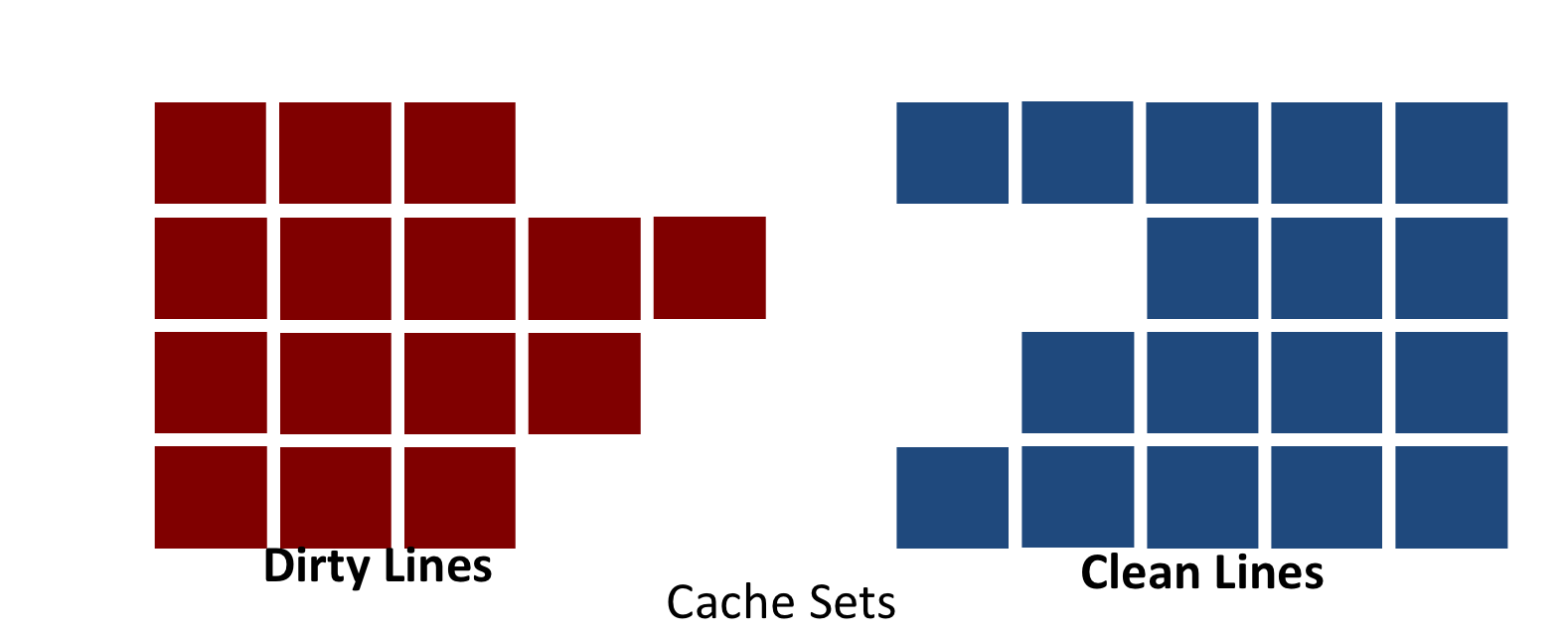
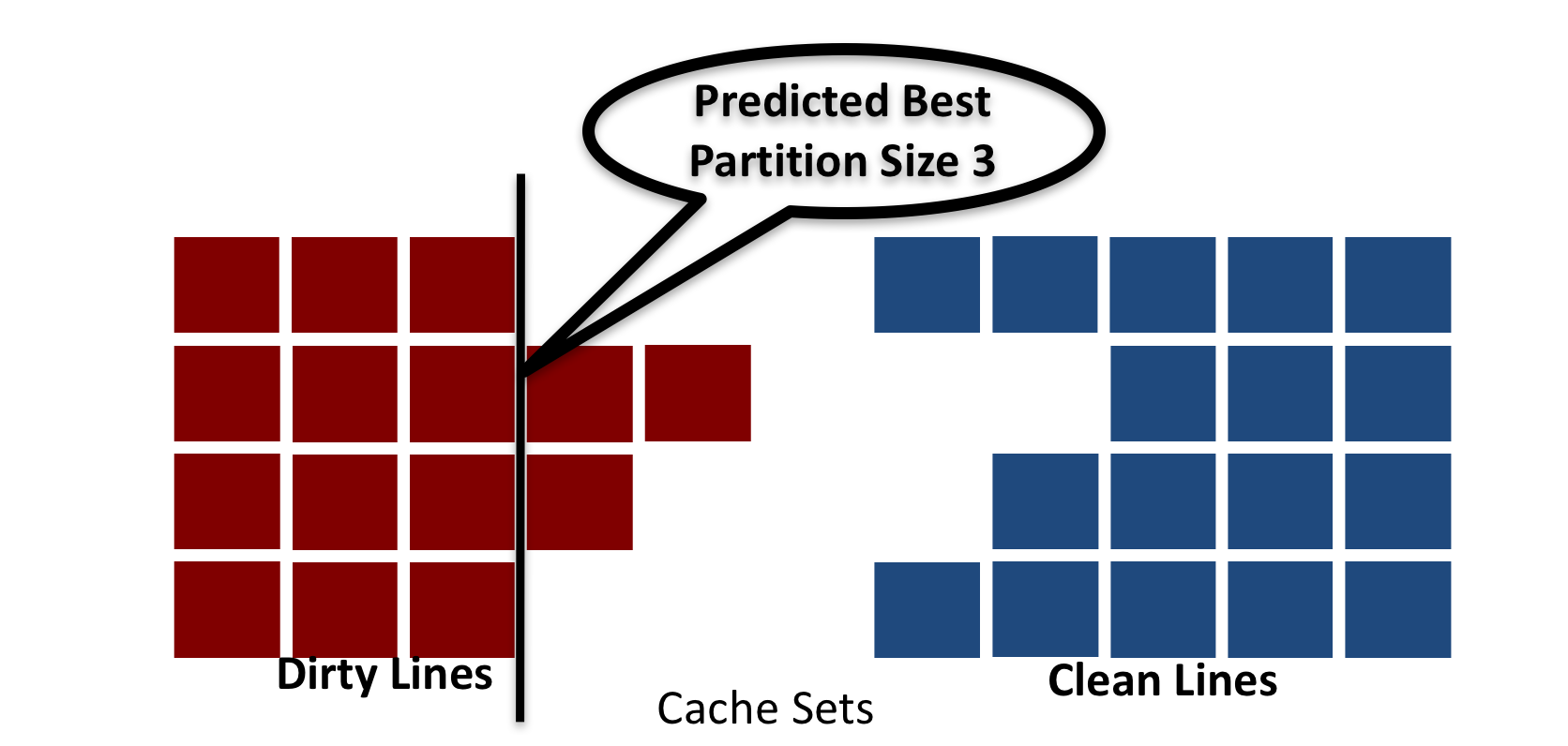
1. Current number of dirty lines is lesser than the predicted the dirty partition size.

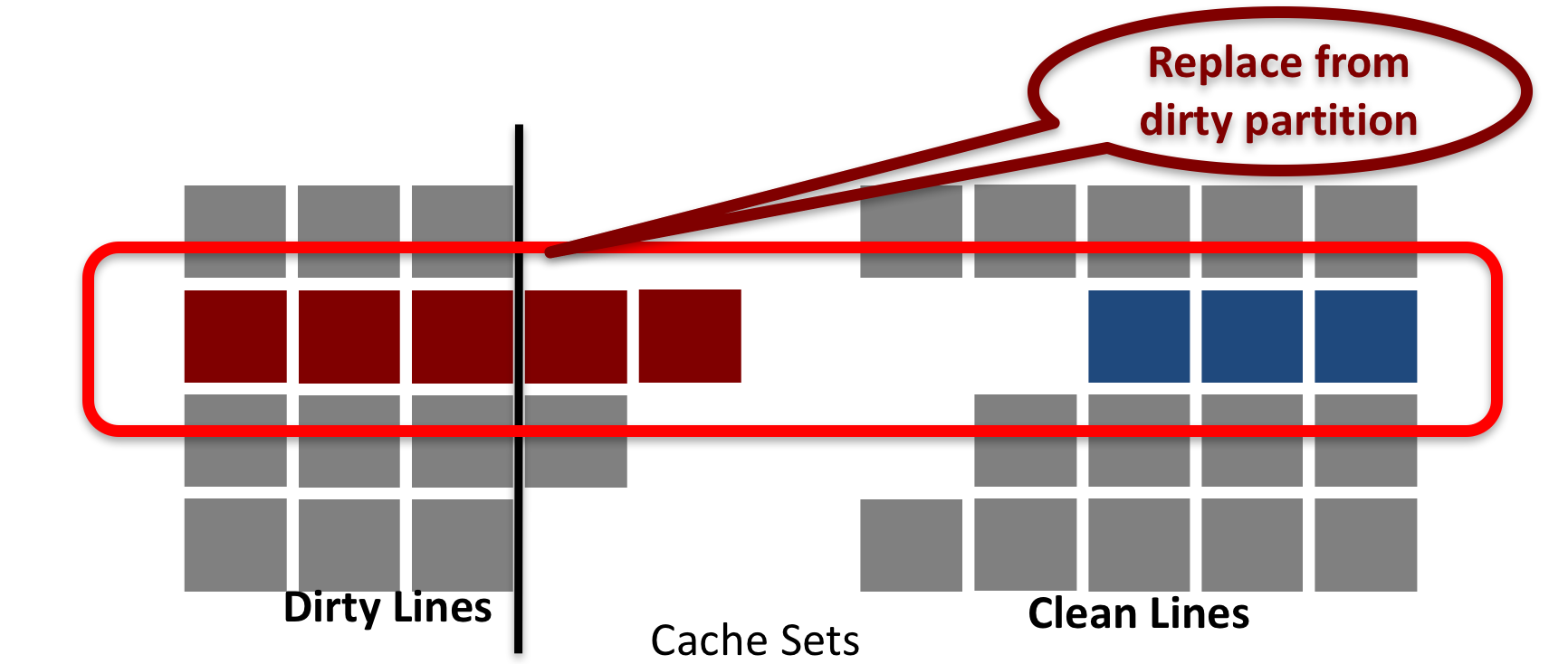
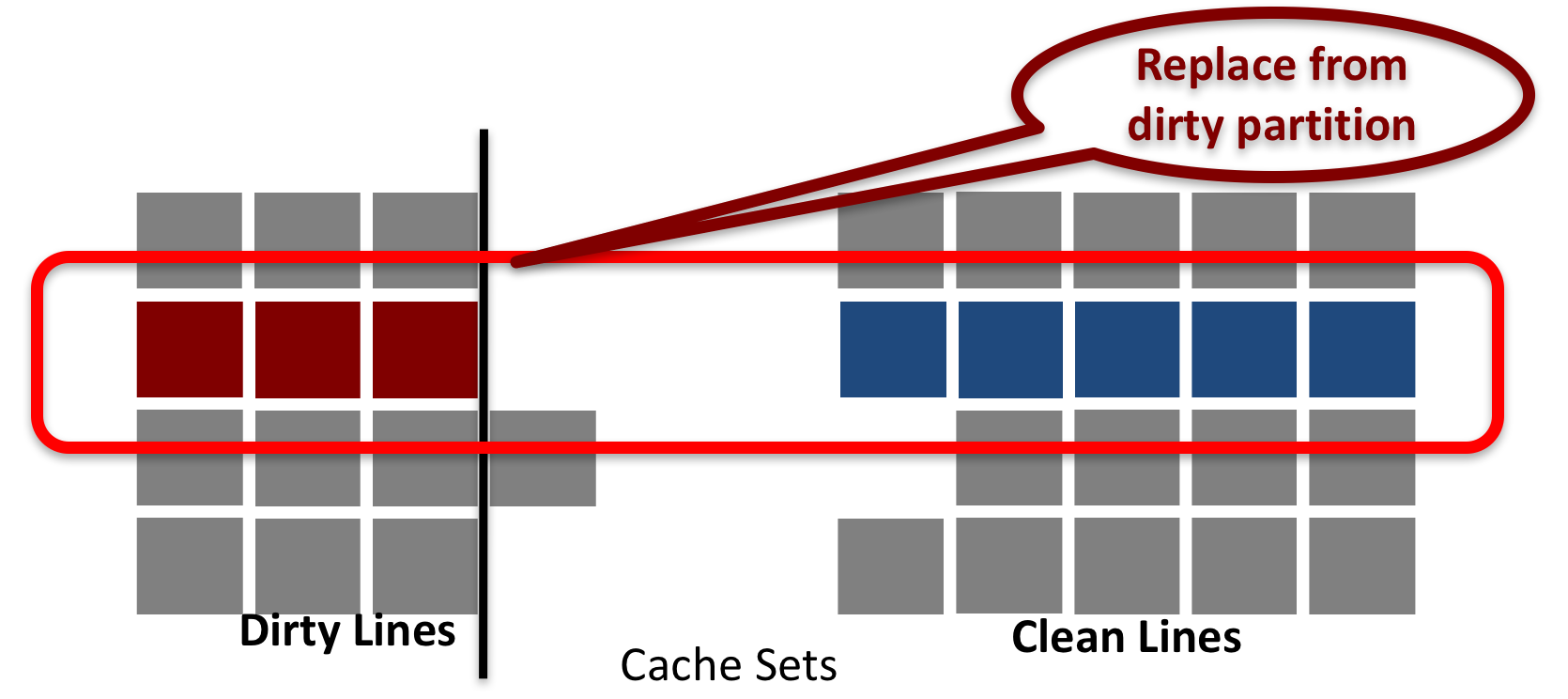
RWP picks the LRU line from the clean partition as the replacement victim.

1. Current number of dirty lines is equal to the predicted the dirty partition size.

If the access is a read, RWP picks the replacement victim from the clean partition. Similarly, a write access triggers a replacement from the dirty partition. The following pictures will explain how the RWP framework will work.







* 1. Predicting partition sizes

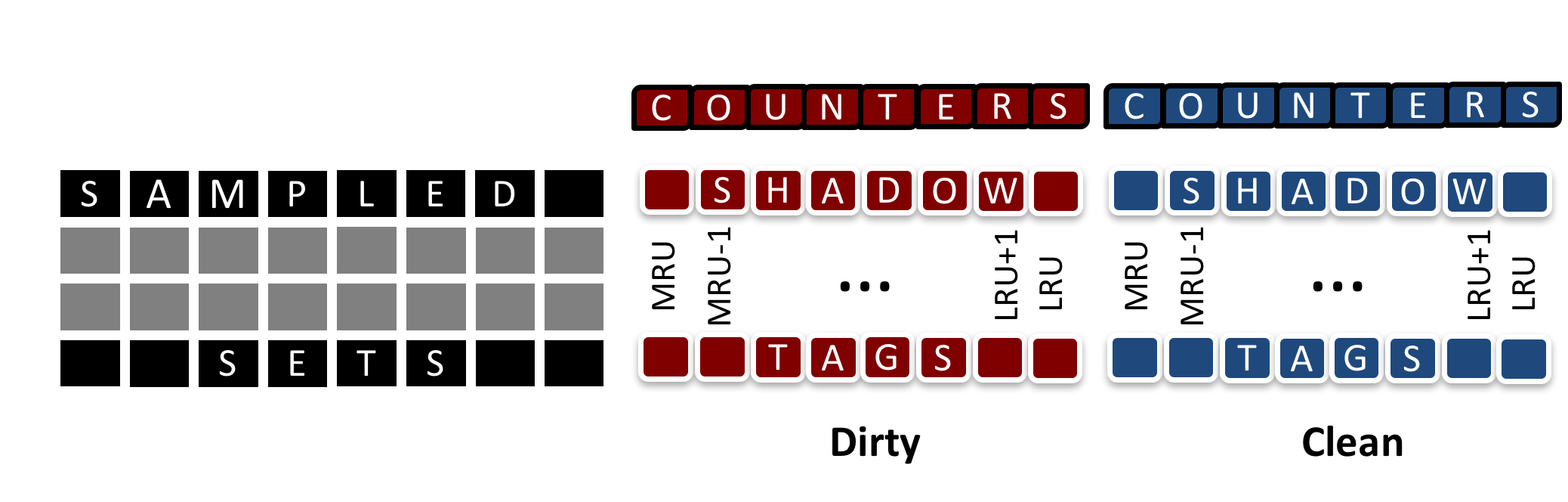
Based on the workload, the partition size predictor should adjust the partition sizes for the clean and dirty lines. RWP assumes that there is exclusive access to cache. RWP employs set sampling, with subset that are called shadow directories. Each sampled subset is divided into two shadow directories: clean shadow directory and dirty shadow directory. Each read miss allocates a line to the clean shadow directory and each write miss allocates a line to the dirty shadow directory. A write request hitting in the clean shadow directory, will put the clean partition to dirty partition. Counters are implemented for maintain the age of the directories for the least and most used cache line. RWP maintains two global age hit counters: a clean age hit counter and a dirty age hit counter. By comparing the values of the dirty and clean age hit counters, our mechanism can predict the number of additional hits and additional misses will happen in the cache. The best partition size is predicted from algorithm based on the paper [3].

The partitioning algorithm reads the hit counters. The utility information in the hit counters directly correlates with the reduction in misses for a given application when given a fixed number of ways.

On a cache miss, the replacement engine counts the number of cache blocks that belong to the miss-causing application in the set. If this number is less than the number of blocks allocated to the application, then the LRU block among all the blocks that do not belong to the application is evicted. Otherwise, the LRU block among all the blocks of the miss-causing application is evicted.

The predictor should be implemented with the following features.

1. Predicts partition size using sampled shadow tags
2. Counts the number of read hits in clean and dirty lines
3. Picks the partition that maximizes number of read hits



The best partition size predicted using the sampled shadow tags. The partition size us determined by the clean and dirty lines. The number of read hits in the shadow tags are counted. The partition is picked up that maximizes the number of read hits.

1. **Evaluation methodology**

Performance evaluation of different cache replacement policies has been done using the simoutorder simulators from the Alpha version of the SimpleScalar toolset. Selected benchmarks from SPEC CPU2000 suite have been used as a simulation workload, representing the state-of-the-art applications for high performance computing. For most evaluation purposes, we have used Split L1 Instruction and Data Cache and Unified L2 Cache. RWP is confined to the LLC, and does not need any additional information from the processor, the L1 or L2 caches.

The experiment is performed based on comparison of LRU vs RWP with changing parameters like associativity and cache size.

* 1. Baseline configuration

The level 1 data cache set of 128, with a block size of 32B 4-way L1 data cache and a 1MB, with a block size of 64B 4-way shared last-level cache. The replacement policy LRU is taken as a baseline.

. /RUNgcc ../../simplesim-3.0/sim-outorder ../../spec2000binaries/gcc00.peak.ev6 -max:inst 50000000 -fastfwd 20000000 -redir:sim sim\_output\_file\_RWP -bpred 2lev -bpred:2lev 1 256 4 0 -bpred:ras 8 -bpred:btb 64 2 -cache:dl1 dl1:128:32:4:l -cache:dl2 ul2:1024:64:4:l

* 1. Benchmarks

We are considering 24 benchmarks from the SPEC CPU2000 suite for the project. The benchmarks need to be analyzed for read and write intensive workloads for effective analyses of the project. By getting the load store instruction counts in the simplescalar toolset, the benchmarks were analyzed for read/write intensiveness.

Memory Trace code for dumping the load and store instructions in a benchmark is added to the simplescalar simulator (Memory trace code was available from a github account [6]).

In the below collected data the benchmarks can be divided into two categories read intensive or write intensive. The benchmarks swim and lucas are the most read intensive benchmarks with more than 90 percent load intensive instructions. The benchmarks parser, mgrid, fma3d, crafty, and ammp have almost 80 percent load intensive instructions. The benchmarks vpr, twolf, mesa, gzip, gcc, equake, bzip2, art and applu are 70 percent load intensive instructions. The benchmarks vortex, mcf, gap, galgel, are equally load and store intensive instruction set benchmarks. apsi is the only benchmark which is store intensive instruction set.

As we demonstrate in this section, different workloads exhibit different types of behavior and different mixes of write- only, dirty-read, and clean lines with different criticality. It is very important to take the behavior of each benchmark to effectively understand the working of the algorithm.

* 1. Simulated Cache Configurations

Simplescalar was simulated with the implementation of RWP and the results were compared between LRU and RWP with various cache configurations by changing compiler commands for cache size and associativity. Please refer to APPENDIX A to see the implementation in simplescalar simulator.

Baseline: We use LRU as our baseline policy.

Dynamic Insertion Policy (DIP) in RWP: DIP is an insertion policy that can dynamically choose to insert cache lines either at MRU or LRU position to eliminate cache thrashing. It protects a portion of the working set of a thrashing workload by inserting lines at the LRU position in the LRU stack, and protects recently-used lines in LRU-friendly workloads by inserting lines at the MRU position.

1. **Results /performance analysis**

The RWP is implemented in Simple scalar and only in single core environment as the simulator puts a restriction to it. Various metrics like miss rate, replacement rate, AMAT, Memory traffic and storage overhead are discussed in this section.

* 1. Miss rate

For any cache management policy miss rate is a very good metric to understand the performance of the policy. Miss rate is the ratio of number of misses to total number of access in the cache. Less the miss rate, more the success of the cache management as the hit rate increases. In the below figure, we show the comparison of miss rate of RWP with the baseline. Our mechanisms reduce miss rates for most benchmarks.

* The read intensive instruction benchmark swim and lucas have a reduction of miss rate by 4.4% and 10%. This really shows the RWP implementation correctness in simplescalar for the project.
* The benchmarks with 80 % read intensive instructions like parser, mgrid, fma3d, and ammp have a reduction of miss rate from 10% to 2.5 %.
* The benchmarks with 70 % read intensive instructions like vpr, twolf, mesa, gzip, equake, bzip2, art and applu have a reduction of miss rate from 2 % to 5 %.
* The benchmarks vortex, mcf, gap, galgel, have equal read and write intensive instructions and hence give no performance difference.
* Apsi is the most write intensive instruction benchmark and it has miss rate reduction of 0.08 %.
* Crafty, gcc and gzip give worse results.

From the above discussion, it is evident that there is a trend. These trends confirm that the RWP is effective in a Read intensive instruction set application and performs least with a write intensive instruction set application. The above data is for cache set of 1024 , block size of 64 B and 4-way associativity.

The trend of miss rate for increasing the cache size was also analyzed. As the cache size increases the miss rate reduces as there is more cache lines to occupy. The cache set was varied from 1K, 2K, 4K and 8K.

* The read intensive instruction benchmark swim and lucas have a reduction of miss rate by 9 % over the increase in cache size
* The benchmarks with 80 % read intensive instructions like parser has a reduction of miss rate by 57 % over the increase in cache size.
* The benchmarks with 70 % read intensive instructions like vpr, bzip2 and applu have a reduction of miss rate by 34 % over the increase in cache size.
* The benchmarks mcf, have equal read and write intensive instructions and hence give no difference in the miss rate.

With cache size changing and keeping the other two parameters constant only compulsory miss will have effect or may be the same, capacity misses will not happen as there are more room for the data, the conflict misses are unchanged.

The trend of miss rate for increasing the cache associativity was also analyzed. As the cache size increases the miss rate will not have change. We selected the benchmark mcf which has 60 % read intensive instructions and 40 % write intensive instructions. As we can see in the below graph, only for associativity of 32 there is an increase in miss rate and there is no change of associativity in miss rates.

* 1. Replacement rate

The most important metric to be calculated with any new cache eviction policy is to compare the eviction policy's replacement rate as compared with the traditional replacement policies. Replacement rate is the ratio of number of replacements to total number of access in the cache. In the below figure, we show the comparison of replacement rate of RWP with the baseline. Our mechanisms reduce replacement rates for most benchmarks. This section will show the effectiveness of the DIP algorithm implemented in the simulator.

* The read intensive instruction benchmark swim and lucas have a reduction of miss rate by 4.4% and 10%. This really shows the RWP implementation correctness in simplescalar for the project.
* The benchmarks with 80 % read intensive instructions like parser, mgrid, fma3d, and ammp have a reduction of miss rate from 10% to 2.5 %.
* The benchmarks with 70 % read intensive instructions like vpr, twolf, mesa, equake, bzip2, art and applu have a reduction of miss rate from 2 % to 5 %.
* The benchmarks vortex, mcf, gap, galgel, have equal read and write intensive instructions and hence give no performance difference.
* Apsi is the most write intensive instruction benchmark and it has miss rate reduction of 0.08 %.
* Crafty, gcc and gzip give worse results.

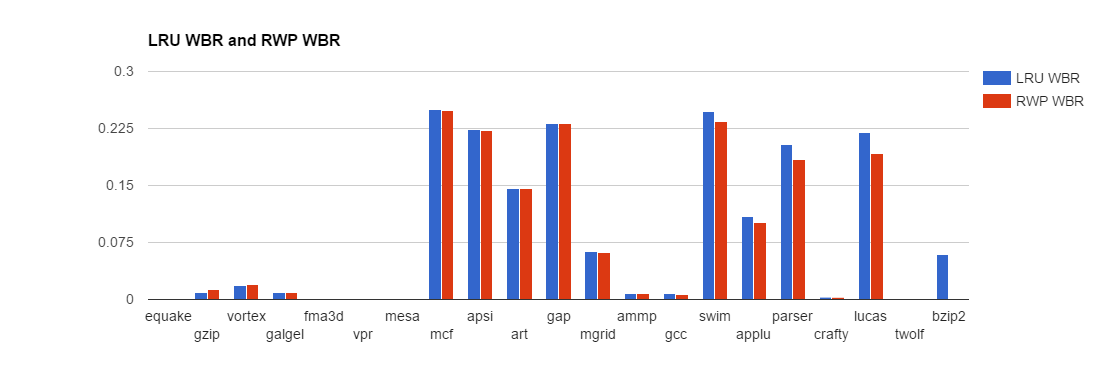
The trend of replacement rate for increasing the cache size was also analyzed. As the cache size increases the replacement rate reduces. The cache size was varied from 1K, 2K, 4K and 8K.

* The read intensive instruction benchmark swim and lucas have a reduction of replacement rate by 70 % over the increase in cache size
* The benchmarks with 80 % read intensive instructions like parser has a reduction of replacement rate by 80 % over the increase in cache size.
* The benchmarks with 70 % read intensive instructions like vpr, bzip2 and applu have a reduction of replacement rate by 30 % over the increase in cache size.
* The benchmarks mcf, have equal read and write intensive instructions and give about 1 percent decrease in the replacement rate.

The trend of replacement rate for increasing the cache associativity was also analyzed. As the cache size increases the replacement rate will not have change. We selected the benchmark mcf which has 60 % read intensive instructions and 40 % write intensive instructions. As we can see in the below graph, only for associativity of 32 there is an increase in replacement rate and the replacement rate decreases for linearly with increase in associativity.

* 1. Memory traffic

The RWP policy improves over the baseline LRU in terms of the write back to the memory and thus help to reduce the memory traffic. This also implies that the power consumption will also decrease. But as the Simple-Scalar does not support power calculation we are not able to confirm it empirically. The below chart shows the comparison of RWP write back rate compared to the LRU write back rate for all the SPEC2000 benchmarks.



For almost all benchmarks we are seeing improvement except few outliers as explained below:

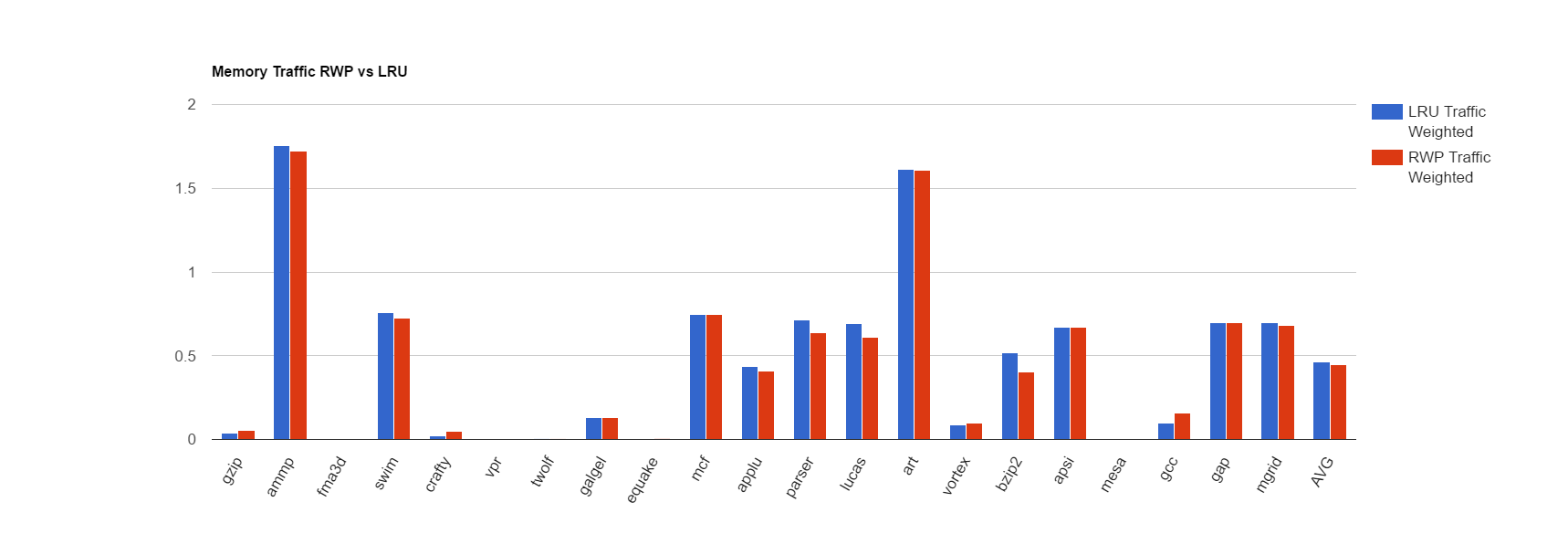
* For bzip2 and twolf we are observing improvements of 99% and 50% in write back rate compared to LRU policy. These both benchmarks, being read intensive, benefit hugely from the read-write disparity.
* For gcc, swim, applu, parser, crafty, and lucas we are observing improvement in range of 5%-12% in write back rate compared to LRU policy. The gcc and crafty are having more misses with RWP compared to the LRU but at the same time has less replacement rate with RWP compared to LRU. It indicates that both these benchmarks are more read intensive and tend to experience more read misses on the clean partition rather than the dirty partition.
* For the fma3d, vpr, mesa, mcf, apsi, art, mgrid, gap and ammp we are observing very less improvements to no improvement over the LRU policy.
* The outliers here are equake, gzip, vortex and galgel where we are observing that the LRU is better. This indicates that in these benchmarks more replacements take place in the dirty partition and so write backs increase in large amount as shown by the chart.

The trend of write backs to memory for increasing the cache size was also analyzed. As the cache size increases the write backs. The cache size was varied from 1K, 2K, 4K and 8K.

* The read intensive instruction benchmark swim and lucas have a reduction of replacement rate by 70 % over the increase in cache size
* The benchmarks with 80 % read intensive instructions like parser has a reduction of write backs by 80 % over the increase in cache size.
* The benchmarks with 70 % read intensive instructions like vpr, bzip2 and applu have a reduction of write backs by 30 % over the increase in cache size.
* The benchmarks mcf, have equal read and write intensive instructions and give about 1.6% percent decrease in the write backs.

The trend of write backs for increasing the cache associativity was also analyzed. As the cache size increases the write backs will not have change. We selected the benchmark mcf which has 60 % read intensive instructions and 40 % write intensive instructions. As we can see in the below graph, the writebacks linearly decreases with increase in associativity.

To quantify the improvement in memory traffic we have combined the misses, replacements and the write backs, weighted by the total memory references/access and compared for all the SPEC2000 benchmarks with the LRU policy and we can see that the overall RWP performs better than the LRU and has reduced memory traffic. It is evident from the below chart:



For almost we are seeing improvement except few outliers as explained below:

* The main benefit of the RWP is reaped by bzip2 with memory traffic reduction upto 22.47%. The benchmarks parser, lucas, and vpr the improvement is in the range of 10-12%. From this is it is evident that the dynamic partition based on read-write disparity can resolve the misses and thus reduce the replacements/write backs.
* We see a similar observation for the benchmarks, applu, swim, twolf, mgrid, ammp, mesa, galgel, gap, art, apsi and mcf, but the improvement is in range upto 6%.
* The outliers in this case are also craft, gcc, gzip, equake, vortex and fma3d. Their memory traffic is increasing owing to increase in the miss rate, replacement rate and write back rates as mentioned in the previous sections.
* Overall, we can see an improvement in the average memory traffic of about 3% with outliers considered as part of average calculation.
  1. AMAT

RWP decreases the miss rate in the cache. Average memory access time also decreases with the miss rate. It focuses on how [locality](https://en.wikipedia.org/wiki/Locality_of_reference) and [cache](https://en.wikipedia.org/wiki/Cache_(computing)) misses affect overall performance and allows for a quick analysis of different cache design techniques.

The AMAT is given by the following equation:

AMAT = Hit time + (Miss Rate X Miss Penalty)

Thus, reducing the memory access time, increases overall performance. CPU-Memory gap is a major performance obstacle.

* 1. Storage overhead

Auxiliary tag directories: We require two ATDs, one each for clean and dirty partition, for each set. The size of each of these directories is equal to associativity, so to sum for a M number of sets and N associativity we will require:

Memory = N\*M\*2

Global Counters: We will two global counters, one each for clean and dirty partition, that will be shared by all the sets. The size of each of these counters equal to associativity. The memory required will be:

Memory= N\*2

The number of counters needed will be equal to the sampled set number.

Core cache overhead: RWP does not keep information in L1 and L2 caches and does not involve PC based prediction. This reduces the cost significantly compared to the other PC based training predictor.

* 1. Summary

In this section, we showed that our RWP outperform the baseline LRU. An advantage of RWP is that it is confined to the last-level cache, and does not require any information from the processor or higher-level caches. RWP outperforms LRU by a reduction of 10% - 0.08 % in miss rate, reduction of 10% - 0.1 % in replacement rate and reduces memory traffic by 50% - 12% for single core using the simplescalar simulator with CPU2000 suite benchmarks. Our evaluations show that RWP is simple and cost effective way to implement and achieve cache performance.

1. **Conclusions and future work**

In this project, we have analyzed the paper [1] well and implemented the RWP and contrasts over the baseline version. We have implemented all the subtasks that we included in our project proposal. The paper exploited the observation that cache read requests are frequently on the critical path whereas cache writes are not. This improved the system performance and resulted in a better cache management. RWP mechanism dynamically partitions the cache into clean and dirty partitions to maximize the read hit. It protects the partition that receives more read re- quests by evicting lines from the other partition that exhibits less read reuse. For a single-core system, RWP with DIP outperforms LRU by a reduction of 10% - 0.08 % in miss rate, reduction of 10% - 0.1 % in replacement rate and reduces memory traffic by 50% - 12%. We have achieved performance increase as we mentioned in objective.

The project can also be extended to have other cache configurations like RRIP and SUP.RRIP is another insertion policy that avoids thrashing and scanning by inserting cache lines either at the end of the recency stack or the middle of the recency stack. SUP uses a PC-based predictor to identify cache lines that are referenced only once.

However, in the paper [1], the key idea being exploiting the read write disparity, can draw many future ground breaking works. The simulator simple scalar is a single core environment and the effect of RWP as a potential cache management policy could not be explored. Also as the paper suggests, multi-threaded workloads can have different reaction to the RWP policy. This can help to understand systems that have different requirements for applications to run in different threads. Write operations can be critical for non-volatile memory and this paper surely helps such memory management by providing a different outlook.

1. **Contrasting: PAPER [1] and PROJECT**

|  |  |  |
| --- | --- | --- |
|  | **Paper [1]** | **Project** |
| **Simulator environment** | CMP$im, a Pin-based [20] x86 simulator | simoutorder simulators from the Alpha version of the SimpleScalar |
| **Cache configuration** | 32KB, 4-way L1 instruction cache, a 32KB, 8-way L1 data cache, a 256KB 8-way L2 cache, and a 4MB, 16- way shared non-inclusive/non-exclusive last-level cache. | level 1 data cache size of 128, with a block size of 32B 4-way L1 data cache and a 1MB, with a block size of 64B 4-way shared last-level cache |
| **Benchmarks** | SPEC CPU2006 suite  With selected cache-sensitive subset | SPEC CPU2000 suite selected 24 benchmarks and analyzed for the read and write intensive instruction set |
| **Core configuration** | Single core and multi core | Single core |
| **Algorithms Implemented** | RWP and RRIP | RWP with DIP |
| **Results** | Miss rate 15 %  Memory traffic 16 % | Miss rate varying from 10% to 0.08 %  Replacement rate varying from 10% to 0.1 %  Memory traffic varying from 50% - 12%. |

The table above summarizes differences between the experiment methodology of our project with the paper [1]. We have contrasted the environment differences in simulator, cache configuration, Benchmark selections, algorithm implemented and results obtained.

1. **References**
2. Samira Khan, Alaa R. Alameldeen, Chris Wilkerson, OnurMutlu and Daniel A. Jimenez. Improving cache performance using read-write partitioning. HPCA, 2014.
3. M. Qureshi et al. A case for MLP-aware cache replacement. ISCA, 2006.
4. M. Qureshi et al. Utility-based cache partitioning: A low- overhead, high-performance, runtime mechanism to partition shared caches. MICRO, 2006.
5. M. Qureshi et al. Adaptive insertion policies for high performance caching. ISCA, 2007.
6. S. T. Srinivasan et al. Locality vs. criticality. ISCA, 2001.
7. <https://github.com/aivus/SimpleScalar>
8. Burger D., Austin T., “The SimpleScalar Tool Set, Version 2.0,” University of Wisconsin-Madison Technical Report #1342, June 1997.
9. **Appendix A**

**Cache.h**

cache\_set\_t

int no\_of\_dirty\_lines;

int \*clean\_atd;

int \*dirty\_atd;

cache\_t

int \* global\_dirty\_age\_cntr;

int \* global\_clean\_age\_cntr;

int dirty\_line\_predn;

int clean\_line\_predn;

cache\_blk\_t

unsigned int dirty\_bit;

/\* cache replacement policy \*/

enum cache\_policy {

LRU, /\* replace least recently used block (perfect LRU) \*/

Random, /\* replace a random block \*/

FIFO, /\* replace the oldest block in the set \*/

RWP //for RWP

};

**Cache.c**

cache\_create

cp->dirty\_line\_predn=0; //predicted dirty lines

cp->clean\_line\_predn=0; //predicted clean lines

//creating global counters for read and write

cp->global\_dirty\_age\_cntr=(int\*)malloc(sizeof(int)\*(cp->assoc));

cp->global\_clean\_age\_cntr=(int\*)malloc(sizeof(int)\*(cp->assoc));

for(int k1=0;k1<(cp->assoc);k1++)

{

cp->global\_clean\_age\_cntr[k1]=0;

cp->global\_dirty\_age\_cntr[k1]=0;

}

cp->sets[i].no\_of\_dirty\_lines=0;

//creating the shadow directories for the each and every set and intialize to 0 initially

cp->sets[i].clean\_atd=(int \*)malloc(sizeof(int)\*(cp->assoc));

cp->sets[i].dirty\_atd=(int \*)malloc(sizeof(int)\*(cp->assoc));

for(int k1=0;k1<(cp->assoc);k1++)

{

cp->sets[i].clean\_atd[k1]=0;

cp->sets[i].dirty\_atd[k1]=0;

}

/\* parse policy \*/

enum cache\_policy /\* replacement policy enum \*/

cache\_char2policy(char c) /\* replacement policy as a char \*/

{

switch (c) {

case 'l': return LRU;

case 'r': return Random;

case 'f': return FIFO;

case 'w': return RWP; // RWP policy

default: fatal("bogus replacement policy, `%c'", c);

}

}

/\* print cache configuration \*/

void

cache\_config(struct cache\_t \*cp, /\* cache instance \*/

FILE \*stream) /\* output stream \*/

{

fprintf(stream,

"cache: %s: %d sets, %d byte blocks, %d bytes user data/block\n",

cp->name, cp->nsets, cp->bsize, cp->usize);

fprintf(stream,

"cache: %s: %d-way, `%s' replacement policy, write-back\n",

cp->name, cp->assoc,

cp->policy == LRU ? "LRU"

: cp->policy == Random ? "Random"

: cp->policy == FIFO ? "FIFO"

: cp->policy == RWP ? "RWP" //RWP policy

: (abort(), ""));

}

cache\_access

/\* low-associativity cache, linear search the way list \*/

blk\_no=0; // variable to identify which block is running

for (blk=cp->sets[set].way\_head;

blk;

blk=blk->way\_next)

{

if (blk->tag == tag && (blk->status & CACHE\_BLK\_VALID))

goto cache\_hit;

blk\_no++;

}

}

**Miss part**

case RWP: //Policy implementation

for ( int k2=0; k2<cp->nsets; k2++)

{

for(int l1=0;l1<cp->assoc;l1++)

{

cp->global\_dirty\_age\_cntr[l1]+=cp->sets[k2].dirty\_atd[l1];

cp->global\_clean\_age\_cntr[l1]+=cp->sets[k2].clean\_atd[l1];

}

}

int max=0,total\_clean=0,total\_dirty=0;

for(int k=0;k<=cp->assoc;k++)

{

for(int l=0;l<k;l++)

{

total\_dirty+=cp->global\_dirty\_age\_cntr[l];

}

for(int l=k;l<cp->assoc;l++)

{

total\_clean+=cp->global\_clean\_age\_cntr[l];

}

if(total\_clean+total\_dirty>max)

{

max=total\_clean+total\_dirty;

cp->dirty\_line\_predn=k;//changed

}

}

if(cmd==Write)

{

if((cp->dirty\_line\_predn)> cp->sets[set].no\_of\_dirty\_lines)

{

repl=cp->sets[set].way\_tail;

while(repl!=NULL) // If data is not null

{

if(repl->dirty\_bit==0) // if its from read partition then we are good else choose next from the way list

break;

repl=repl->way\_prev;

}

if(repl==NULL) //if null choose next

repl=cp->sets[set].way\_tail;

cp->sets[set].no\_of\_dirty\_lines++;

repl->dirty\_bit=1;

}

else // if Dp<Dc

{

repl=cp->sets[set].way\_tail;

while(repl)

{

if(repl->dirty\_bit==1)

break;

repl=repl->way\_prev; //else choose previous in the list

}

if(repl==NULL) //if null choose next

repl=cp->sets[set].way\_tail;

}

}

else // Read miss case for replacement

{

if((cp->dirty\_line\_predn)< cp->sets[set].no\_of\_dirty\_lines)

{

repl=cp->sets[set].way\_tail;

while(repl)

{ // to check if replacement is from the dirty patition , if not choose from the previous

if(repl->dirty\_bit==1)

break;

repl=repl->way\_prev;

}

if(repl==NULL)//if null choose next

repl=cp->sets[set].way\_tail;

cp->sets[set].no\_of\_dirty\_lines--; // as we are placing the read miss in the dirty partition we will reduce this value

repl->dirty\_bit=0; // and also move it to clean

}

else // Dp> Dc replace from LRU clean partition

{

repl=cp->sets[set].way\_tail;

while(repl) // check if the replacement is form the clean partition only if not

{

if(repl->dirty\_bit==0)

break;

repl=repl->way\_prev;

}

if(repl==NULL)//if null choose next

repl=cp->sets[set].way\_tail;

}

}

update\_way\_list(&cp->sets[set], repl, Head); /\* put to MSU \*/

break;

**hit part**

if(cp->policy==RWP)

{

if(cmd==Write) // if write then increment corresponding atd entry of dirty

cp->sets[set].dirty\_atd[blk\_no]++;

else// if read then increment corresponding atd entry of clean

cp->sets[set].clean\_atd[blk\_no]++;

}

/\* update dirty status \*/

if (cmd == Write) // if write hit then set dirty bit to 1 and if previous it was clean then increment the dirty lines

{

if(blk->dirty\_bit==0)

{

cp->sets[set].no\_of\_dirty\_lines++;

}

blk->dirty\_bit=1;

}

else

{

if(blk->dirty\_bit==1)

cp->sets[set].no\_of\_dirty\_lines--;

}

if (blk->way\_prev && cp->policy == LRU && cp->policy==RWP)

{

/\* move this block to head of the way (MRU) list \*/

update\_way\_list(&cp->sets[set], blk, Head);

}

/\* update dirty status \*/

if (cmd == Write)

{

if(blk->status!=CACHE\_BLK\_DIRTY)

{

cp->sets[set].no\_of\_dirty\_lines++;

}

blk->status |= CACHE\_BLK\_DIRTY;

}

else

{

if(blk->status==CACHE\_BLK\_DIRTY)

cp->sets[set].no\_of\_dirty\_lines--;

}

1. **Appendix B**

The link to the excel document for the graphs and data accumulated for this project:

